数据库

SQL：结构化查询语言，用于与数据库通讯，

可以创建 1.数据库 2.数据库中的表 3.存储过程 4.图表

DBMS(数据库管理系统)：执行 SQL 查询的软件

SQL vs MySQL：SQL 是一个产品语言，和数据库交流

MySQL 不是语言，是数据库管理系统

inner Join：两张表同时满足的记录

left join：将左表所有查询信息列出来，而右表只列出 On 后条件与左表满足的部分，左表有右表没有的列值为空

right join：将右表所有查询信息列出来，而左表只列出 On 后条件与右表满足的部分，右表有左表没有的列值为空—NULL

MySQL join 原理—nested-loop join

1.简单 nested-loop，取主表(驱动表)每一行找右表每一行去匹配，访问次数多
2.索引 nested-loop 要求非驱动表有索引列，驱动表找到索引后才会回表查询
3.阻塞 nested-loop 把驱动表和 join 相关的列(包括 on 和 select 的列)先缓存到 join buffer 中再去匹配，这样就把第一种 loop 的多次比较合并为一次，效率高

MySQL 执行计划：用 explain 语法来进行查询分析 + navicat 工具

1. SQL 语句执行顺序/执行过程：
from--->join, on--->where--->group by, having--->select, distinct--->order by, limit

2. SQL 一次执行的过程：

1.发 SQL 语句：客户端和 MySQL 服务器建立 TCP 连接后发送一条 SQL 语句给 MySQL 服务器(同一时刻只有一方能发数据，同时，另一方要么完整地接受数据，要么直接断开连接)
2.检查查询缓存：MySQL 检查 SQL 语句是不是 Select 型，是的话去检查查询缓存，缓存命中就返回结果否则去数据库找
3.生成执行计划：MySQL 服务器解析 SQL+预处理(看 SQL 语句有没有错)，然后优化器会预测几个执行计划(指令树)的成本(根据表和索引页数)，选成本最小的那个，不考虑缓存，假设读取都是读磁盘
4.执行：MySQL 遍历指令树，而且调用存储引擎 API 来执行查询
5.返回结果：MySQL 把结果返回给客户端(逐步返回)—结果集每行都会用通信协议封包，然后再通过 TCP 传输

2. **主键**唯一标识一条数据　　不能重复，保证数据完整 一种特殊的唯一索引

外键用来和其他表建立联系 可以重复，是另一个表的主键

3. **索引**(B+树)：一种数据结构，能用它来加快查找数据库数据

没有索引的话，数据按顺序一条条在磁盘上存储(加了主键后，比如自增主键，数据在磁盘

变成了平衡树结构，也就是聚簇索引(表排列顺序和索引顺序一致))

索引**优点**：1.创建唯一性索引能保证数据库表中**每一行数据的唯一性**

2.可能可以加快数据检索和**表与表连接速度**

索引**缺点**：1. 会**减慢写入速度+增删改慢**(因为每次写入时都要更新索引)

2.创建索引和维护索引要**消耗时间**，尤其数据量大的时候

3.**占**物理**空间**和数据空间(数据表)—磁盘空间

索引**使用场景/原则**：1.加在经常需要搜索的列上，不要加在写入多和读取少的列，因为会先访问索引, 每次写入时都要更新索引，执行速度变慢

2.数据多+字段值相同的时候用唯一索引(支持 NULL)

3.字段多+没重复的时候用复合索引(不支持 NULL)

4.用唯一性索引(为经常要查询的建索引，限制索引数目)

5.尽量用最左前缀匹配原则，因为如果索引的值很长，比如 text 和 blog 类型的字段，来全文检索的话会浪费时间，如果只检索字段前面的几个字符，能提高索引检索速度

6.where group by order by 的子句要建索引 count max 也要

什么时候不用：1.要取表里所有记录 2.对不是唯一而且重复的字段，比如姓名 3.经常修改和删除的字段 4.记录比较少的表

索引类型：

1. **单列索引**：只有一个字段的索引 2.**复合索引**：包含两个或以上字段的索引

复合索引只要第一个列名对应后面就都能查到，中间 Y 不写也没事

```
CREATE INDEX 索引名 ON 表名(列名 X, 列名 Y, 列名 Z);
```

建了 3 个索引：

1.单列索引(列 X）2.复合索引(列 X, 列 Y) 3.复合索引(列 X，列 Y，列 Z）

3.唯一索引：表上 一个或者多个字段组合起来建立的索引

字段的值组合起来不可以重复 唯一索引允许有多个 一般只有一个 允许 null

主键：一种特殊的唯一索引 区别：不允许 null 可有可无的

4.聚簇索引—InnoDB 用

1.包含主键索引和对应的数据

2.主键顺序是数据的物理存储顺序，叶子节点是数据节点

3.因为真实数据的物理顺序只有一种，所以一个表最多一个聚簇索引

缺点：如果一个表没有聚簇索引，创建的时候就会对数据重新进行排序，所以它对表的修改速度比较慢，所以不建议在经常更新的列建这个索引

优点：性能好，因为找到第一个索引值，有连续索引值的记录也会在他后面，因为物理上顺序就是在他后面

什么时候用：按照表最常用的 SQL 查询方式来选字段作为聚簇索引

聚簇索引默认是主键，如果表里没有定义主键 InnoDB 会选一个没有 null 值的列代替，如果没有这样的索引，InnoDB 会隐式地定义一个主键作为聚簇索引

聚簇索引 vs 唯一索引：聚簇索引的索引值没有被要求是唯一的，就是在有聚簇索引的列上可以插多个相同的值，这些值在硬盘上的物理排序和在聚簇索的引排序一样

非聚簇索引—MyISAM 用

数据和索引存在不同地方，叶节点仍然是索引节点(存的是主键值)而且是有指向对应数据块的指针

具体怎么用：Mysiam 用 key_buffer 把索引先缓存到内存，当通过索引访问数据的时候，在内存中直接搜索索引，然后通过索引找到磁盘里对应的数据，如果索引不在 key_buffer 命中时，速度就会慢

聚簇 vs 非聚簇区别：聚簇索引能直接查到数据，非聚簇索引先查到主键值，再用主键值查数据

非聚簇索引都是辅助索引，像复合索引、前缀索引、唯一索引一样，辅助索引叶子节点存储的不是行的物理位置，而是主键值

覆盖索引：不用聚簇索引就能直接查到数据，前提是符合最左侧匹配原则

例子：建立索引 createindex index_birthday on user_info(birthday);//查询生日在 1991 年 11 月 1 日出生用户的用户名 select user_name from user_info where birthday ='1991-11-1'

1.通过非聚集索引 index_birthday 查找 birthday 等于 1991-11-1 主键值

2.用主键执行聚集索引找到数据（数据行）存储的位置

3.从得到的真实数据中取得 user_name 字段的值

索引失效：1.列与列对比 2.条件里有 or 3.like 查询以%开头

4.索引列没有限制 not null 5.如果列类型是字符串，没有在条件中 把数据用引号引用起来

创建索引：1.普通索引 CREATE INDEX indexName ON mytable(username(length))

ALTER TABLE tbl_name ADD INDEX index_name (column_list)

2.唯一索引

CREATE UNIQUE INDEX indexName ON mytable(username(length))

3.主键索引 添加 PRIMARY KEY

ALTER TABLE `table_name` ADD PRIMARY KEY (`column`)

4.在创建表的时候加

MySQL 和 MongoDB

MySQL：关系型数据库 B+索引和哈希索引(memory 引擎支持，但是重启后数据会丢失)

MongoDB：非关系型数据库(NoSQL), 用的 bson 格式--适合文档存储和查询

语句：

db.collection.find(); db.collection.distinct("name"); db.collection.find({"age": 22});

db.collection.find({ "field" : { $gte: value } }); // greater than or equal to : field >= value

```
db.collection.update( criteria, objNew, upsert, multi )

criteria : update的查询条件，类似sql update查询内where后面的
objNew  : update的对象和一些更新的操作符（如$,$inc...）等，也可以理解为sql update查询内set后面的
upsert  : 这个参数的意思是，如果不存在update的记录，是否插入objNew,true为插入，默认是false，不插入。
multi   : mongodb默认是false,只更新找到的第一条记录，如果这个参数为true,就把按条件查出来多条记录全部更新。

例:
db.test0.update( { "count" : { $gt : 1 } } , { $set : { "test2" : "OK"} } ); 只更新了第一条记录
db.test0.update( { "count" : { $gt : 3 } } , { $set : { "test2" : "OK"} },false,true ); 全更新了
db.test0.update( { "count" : { $gt : 4 } } , { $set : { "test5" : "OK"} },true,false ); 只加进去了第一条
db.test0.update( { "count" : { $gt : 5 } } , { $set : { "test5" : "OK"} },true,true ); 全加进去了
db.test0.update( { "count" : { $gt : 15 } } , { $inc : { "count" : 1} },false,true );全更新了
db.test0.update( { "count" : { $gt : 10 } } , { $inc : { "count" : 1} },false,false );只更新了第一条
```

删除$pull : { field : value } } $表示自己　insert 表示加

MySQL: 坏处：用表存数据在磁盘(读取比较慢) 定义表和字段才能存

好处：有 Join 来进行复杂查询，可以事务处理，可以来保证数据的一致性

NoSQL: 坏处：没有 join，不支持事务，不提供附加功能，比如报表

好处：基于文档设计，把数据放内存，比在磁盘读取快，满足高并发

即使放内存，也比 SQL 简单，因为它是半结构化的数据格式，MySQL 的存储是经过结构化、多范式有很多复杂规则的数据，还原内存结构慢

Redis：缓存数据库，用于存储使用频繁的数据到缓存，读取速度快

而 MySQL 用于持久化地存数据到硬盘，速度慢

Redis 是一个 key-value 存储系统，value 有很多类型而且有 add/remove, push/pop 操作，都是原子性的

数据怎么存放？数据被放在 Mysql 设计的**数据页**上，数据页上数据是**一行行**的，格式是 **compact**，每行都有**行描述**和**指针**去指下一行

行式存储(MySQL) vs 列式存储(Hbase)

1.行式存储倾向于结构固定，列式存储倾向于结构弱化

2.行式存储存储一行数据只要一个主键，列式存储要多个主键

3. 行式存储要维护大量索引和物化视图，所以在时间(处理)和空间(存储)上成本都很高

列式存储只访问查询涉及的列，能很大地降低系统 I/O

数据库事务：DBMS 执行过程中的一个**逻辑单位**，由一个数据库**操作序列**组成

目的：1.提供操作序列从失败到恢复正常的**方法**，同时让数据库在异常情况下仍然能**保持一致性**

2.多个应用程序并发访问数据库的时候**提供一个隔离**，让程序的操作不会干扰到别的操作

4 个**特性**：1.原子性(所有操作要么全都执行成功，要不全部取消)

由重做日志实现

2.一致性(事务开始前和结束后，数据库完整性没有被破坏)

由回滚日志实现

3.隔离性(多个事务并发执行，一个事务的执行不影响其他事务的执行，也就是事务提交之前对其他事务不可见)

由数据库的锁实现

4.持久性(事务提交后，它的结果是永久的) 由重做日志实现

隔离等级：1.读未提交(一个事务可以读其他还没提交的事务的执行结果)

2.不可重复读(一个事务只能读到别的事务已经提交的内容)

3.可重复读(MySQL 的等级)确保一个事务在重复读取数据时候，对重复读取到的数据行的值是不变的

4. 串行化(事务一个个排队执行，只有当事务提交后，其他事务才能从数据库中查看数据的变化) 执行效率很差，可能导致锁竞争

3 种需要阻止的现象：

脏读：一个事务**读到**别的事务写入但还**没提交的数据**

不可重复读：两次查询的内容不一样，因为期间别的事务做修改了

幻读：事务按照之前条件重新查询时，结果集的个数不一致，然后多出来的一行是幻行，也就是读到了别的事务插入的数据

MySQL 解决幻读：1.MVCC(快照读/当前读)：将历史数据**存一份快照**，其他事务**修改数据**是对当前事务**不可见**的

快照：select*from where

当前读：update

2.next-key lock(当前读)：将当前数据行和**上下两行**的间隙**锁定**，

查哪些数据行就锁住这些行，保证这个范围内读取的**数据**是**一致**的，但并不完全等于串行化的隔离级别

事务隔离级别	脏读	不可重复读	幻读
读未提交（read-uncommitted）	是	是	是
不可重复读（read-committed）	否	是	是
可重复读（repeatable-read）	否	否	是
串行化（serializable	否	否	否

事务**处理命令**:初始化(start)回滚(rollback)提交(commit)

设保存点(savepoint identifier)

MVCC：每个读操作都会看到一个有一致性的 snapshot，而且可以实现非阻塞的读，允许

数据有多个版本，同一时间不同事务看到数据不同
Mysql 的 Innodb 实现 mvcc：给每行加两个字段，分别是这行创建和删除的版本
select：要满足两个条件，Innodb 才会返回当前行的数据：
1.这行的创建版本号小于等于当前版本号，这样保证在 select 操作之前所有的操作已经执行落地
2. 这行的删除版本号大于当前版本或者为空，大于意味着有一个并发事务把这行删除了
insert：把新插的行的创建版本号设为当前系统的版本号
delete：把要删除的行的删除版本号设为当前系统的版本号
update：不执行原地 update，而是转换成 insert + delete。把旧行的删除版本号设为当前版本号，把新行插入然后同时把创建版本号设为当前版本号
同时，写操作（insert、delete 和 update)执行时，要把系统版本号递增

Mysql 事务分类：

1.扁平事务(最简单但最频繁)：begin work 开始到 commit work/rollback work 结束
操作要么都执行要么都回滚
所以是有原子性的应用程序的基本模块
缺点：因为某些地方出错但不会导致所有操作无效的时候还是要回滚之前所有操作, 代价太大，因为它只被隐式地设置了一个保存点
2.带有保存点的扁平事务：支持扁平事务的操作+允许事务执行的时候回滚到事务较早的状态，保存点用来通知事务系统应该记住事务当前状态
3.链事务：是保存点事务的变种，在提交事务的时候会释放不需要的对象，把上下文传给下个要开始的事务，这个是一个原子操作
保存点事务的保存点在系统崩溃的时候会丢失，系统恢复后事务要重新执行而不是从最近保存点继续执行
区别：保存点事务能回滚到任意保存点但链事务只能回滚到当前保存点
4.嵌套事务：是一个层次结构框架，有一个顶层事务控制各个层次的子事务
父事务回滚, 子事务也会被回滚—子事务具有 ACI 特性但没有 D 特性
5.分布式事务：是分布式环境下运行的扁平事务
InnoDB 不支持分布式事务，只支持前 4 个，但它支持 XA 事务，通过 XA 来实现分布式事务，而且这个时候隔离等级必须是串行化
Oracle，SQL server 都支持 XA 事务---由 1.资源管理器(提供访问事务的方法)(通常一个数据库就是一个资源管理器)2.事务管理器(协调全局事务里的各个事务)3.应用程序(定义事务边界)

Mysql 日志分类：

1.重做日志：记录事务执行后的状态，来恢复没写进文件的但已经执行成功的事务它更新的数据　保证事务原子性和持久性
2.回滚日志：保存事务发生前的数据，之后可以用于回滚，还提供 mvcc(多版本并发控制下的读)　　保证事务一致性+数据的原子性
重做日志 vs 回滚日志：回滚日志只是把数据从逻辑上恢复到事务开始前的状态而不是从物理页面上操作实现

3.错误日志：记录 mysql 的启动和停止，还有服务器运行过程中发生的错误
4.普通查询日志：记录了服务器接收到每个查询和命令
5.慢查询日志：记录执行时间过长和没有用索引的查询语句

Mysql 存储引擎：

1.InnoDB：是默认的引擎，支持事务但损失效率 有乐观锁和悲观锁
2.MyISAM ：每个 MyISAM 在磁盘上存 3 个文件 fm(存表定义的数据)MYD(存表具体记录的数据) MYI(存索引) 不支持事务但存储更快(允许读写有错误数据的时候用)
3.Memory 把数据存在内存来提高数据的访问速度，每个表和一个 fm 文件关联
用哈希索引

SQL 优化：1.避免全表扫描，首先考虑在 where 和 order by 涉及的列上建索引
2.避免在 where 子句中进行 null 判断和!=<>，or, in, not in
3.尽可能用 varchar 代表 char，这样能节省存储空间
大表优化：1.限定数据范围(禁止那些不带限制数据范围条件的查询语句)2.读写分离(主库负责写，从库负责读)3.垂直分区(根据表的相关性拆分，也就是把数据库列拆分)4.水平分区(表结构不变，把数据分散到不同的表和库，来达到分布式目的)—最好分库因为分表只是解决单一表数据多的问题，表的数据还是在同一台机器上，不能提升 Mysql 并发能力

数据库三范式(为了设计冗余较小、结构合理的数据库)

1. 第一范式：要保证所有**字段**都是**不可分解的原子值**，也就是有原子性-
比如地址列是浙江省杭州市慧芝湖花园，经常访问地址列的话就要把地址这列分成省份 城市 详细地址三列
2. 第二范式：要保证**每列都和主键有关，一个表只能存一种数据**
比如分开订单编号表和商品编号表
3.第三范式：要保证**每列都和主键直接相关**，不是间接相关，然后是用外键连接不同表去得到别的表的信息

数据库的锁：悲观锁

类型：1.读锁/共享锁(被锁定的对象只允许被读)事务 A 对一个数据上锁，A 只能读不能写，其他事务只能给这个数据加读锁，除非 A 释放锁
2.写锁/排他锁，事务 A 对一个数据加写锁，事务 A 可以读和改数据但其他事务不能读也不能加锁

表级锁：是系统开销最低但并发性最低的一个锁策略，操作对象是数据库
注意：给表加锁要获得所有涉及到表的锁 LOCK TABLE
InnoDB 和 MySIAM 支持 不会产生死锁
什么时候锁住整个表？ 1.磁盘满的时候 2.insert 时候 3.如果对用 InnoDB 的表使用行锁，被锁的字段不是主键，而且也没有对它建索引的话，行锁锁的是整张表
行级锁：操作的对象是表里的一行，开销大 并发性好 SELECT…LOCK IN
InnoDB 支持

页面锁：开销和加锁时间和锁定粒度介于表锁和行锁之间会，出现死锁，并发度也一般
MVCC(多版本并发控制)：处理并发　　开销最大 处理能力最强

死锁：多个资源并发的时候产生。表级锁不会产生死锁
死锁原因：1.事务对资源访问顺序的交替导致(一个用户 A 访问表 A(锁住了表 A)，然后又访问表 B；另外一个用户 B 访问表 B(锁住了表 B)，然后想访问表 A；这样会导致用户 A 因为用户 B 已经锁住了表 B，它就必须等用户 B 释放表 B 才能继续，然后用户 B 要等用户 A 释放表 A 才能继续，这样就产生死锁了)　解决：调整程序的逻辑

2.并发修改同一个记录(用户 A 查一条纪录，然后修改这条纪录；同时用户 B 也想修改这条纪录，这时用户 A 的事务里锁的性质从查询的共享锁想上升到独占锁，但用户 B 里的独占锁因为 A 有共享锁所以必须等 A 释放掉共享锁，而 A 由于 B 的独占锁不能上升独占锁也就不会去释放共享锁，这样就出现了死锁)
解决：乐观锁(实现写-写并发)和悲观锁(保证操作的独占性)

3. 索引不当(2 个情况：1.在事务里执行一条不满足条件的语句导致了全表扫描，把行级锁上升为表级锁，然后多个这样的事务执行后，就很容易产生死锁 2.表中的数据量很大而且索引建的太少或者建的不合适，这样就会经常有全表扫描，这样之后应用系统会越来越慢，最后导致死锁)

解决：SQL 语句中不使用太复杂的关联多表的查询

避免死锁：1.以固定顺序访问表和行 2. 大事务拆小 3.一个事务中一次锁定所有资源 4.降低隔离等级

InnoDB 有自动检查死锁的功能，会自动解决
InnoDB 有三种行锁的算法：
1，Record Lock：单个行上的锁
2，Gap Lock：间隙锁，锁定一个范围，但不包括记录本身
3，Next-Key Lock：1+2，锁定一个范围，而且锁定记录本身。
对于行的查询，都是用这个方法，主要是为了**解决幻读**的问题

B+树：层数一般控制在 3-5 层(IO 次数就是树的高度)，主要为了避免磁盘的读取次数太多，

然后它的数据都在叶子节点，并且叶子节点之间用指针连接，这样区间遍历和查询会更快，然后新的值可以插在已经有的节点里，不用改变树的高度，这样就能减少重新平衡和数据迁移的次数，而且中间节点只保存索引不保存数据所以能容纳更多节点元素

B 树：每个节点都有 key/value, 所以经常访问的节点可能离根节点更近,访问更快
vs B+树：每一层要递归遍历，而且相邻元素可能在内存里不相邻，缓存命中率没有 B+树好，所以 B+树更适合文件系统
缺点：
1、每个节点都有 key 和 data，但是每个页的存储**空间是有限**的，如果 data 比较大的话会导致每个节点存储的 key 数量变小
2、存储的数据量很大的时候会让深度变大，查询时候磁盘 io 次数就多了，这样就会影响查询性能
应用：文件系统和数据库索引

红黑树特点：根节点和叶子节点都是黑色，而且从任一节点到其叶节点黑节点数量一样，二叉树查询复杂度可能是 O(n)--节点都在一边

应用：java 的 treeMap 实现 + IO 多路复用 epoll 的实现 -> 支持快速增删改查

B+/B 树(平衡多路查找树**) vs 红黑树**：

在同一个节点，B/B+树的高度会远小于红黑树，红黑树 IO 次数就更多了

红黑树的索引时间复杂度稳定在 O(logn)，但是读硬盘读取的时候，会导致读取的数据较多（磁盘读取的基本单位是扇区），根据空间局部性原理，需要读的数据的周围数据也可能读取到，所以使用 B+树更好。

AVL 树 vs 红黑树：控制在 O(logn)，但左右子树高度差不能超过 1，所以经常要调整，但红黑树在插入和删除时不会频繁破坏红黑树的规则，不用频繁调整节点位置

应用：Windows NT 内核

SQL 注入：把用户输入数据作为代码执行

1.条件：1.首先用户能控制输入；2.然后是原本程序要执行的代码拼接了用户输入的数据，把数据当代码执行了

2.类型：1.没有正确过滤的字符 2.数据库服务器的漏洞 3.盲目 SQL 注入式攻击 4.条件性差错 5.时间延误

3.注入点类型：1. 数字型 2.字符型 3.搜索型

4.原因：①查询集处理，转义字符处理，多个提交的处理和错误处理不恰当；②数据库配置不安全

5.防御：

1. 用预编译的 SQL 语句，SQL 的语意不会变化，攻击者就不能改变 SQL 的结构了(比如攻击者插入了类似’or ‘1’=’1 的字符串，也只会把这个字符串作为 username 查询)

2.先把 SQL 语句定义在数据库里，从存储过程来防御。(存储过程中可能也存在注入问题，应该尽量避免在存储过程中使用动态 SQL 语句)

3.限制数据类型，统一数据的格式

4.开发时候尽量用安全函数代替不安全函数，写安全的代码(危险函数 C 里的 system())

5.避免高权限用户直接连接数据库

分布式是指通过网络连接的几个组件，然后通过交换信息协作这样 形成的系统

分布式的 CAP 定理：Consistency（一致性）、 Availability（可用性）、Partition tolerance（分

区容错性），最多只能同时三个特性中的两个，三者不可兼得
C：更新操作成功并返回客户端后，所有节点在同一时间的数据完全一致
A：服务一直可用，而且是正常响应时间
P：分布式系统在遇到某节点或网络分区故障的时候，仍然能够对外提供满足一致性或可用性的服务

集群的话是同一种组件的多个实例形成的逻辑上的整体

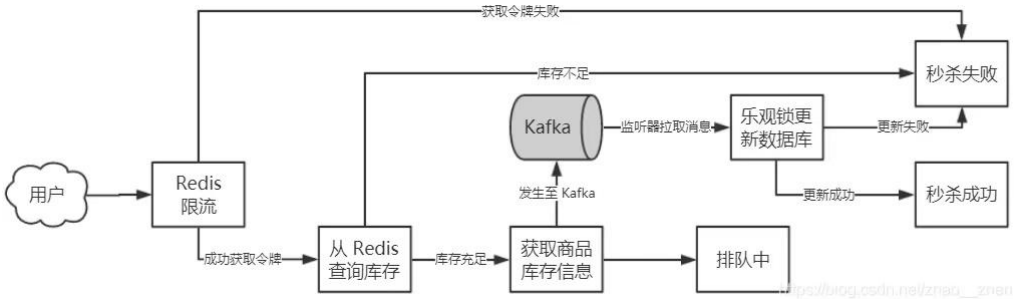
区别：多个不同组件构成的系统就是分布式系统而不是集群
是集群不是分布式系统的情况的话，比如多个经过负载均衡的 HTTP 服务器，它们之间不会互相通信，如果不带上负载均衡的部分的话，一般不叫做分布式系统

微服务是一种架构模式，它就是把单一应用程序划分成一组小的服务，然后服务之间互相配合来给用户提供价值

每个服务运行在他们独立的进程里，服务和服务之间用比较轻量级的通信机制沟通(通常是基于 HTTP 的 RESTful API)。每个服务都围绕业务来构建，而且他们能被**独立地**部署到生产环境。然后要避免的是统一的管理，对于一个服务，要根据业务上下文，选择合适的语言和工具去构建它。

秒杀系统(将请求尽量拦在系统上游同时对请求进行限流和削峰)

特点：1.高性能：因为有大量并发读和写，所以要支持高并发访问。2.一致性：有限数量的库存同时被很多请求同时来减库存，在大并发更新的过程中要保证数据的准确性。
3.高可用：秒杀时会在一瞬间涌入大量流量，为了避免系统宕机，要做好流量控制。
前端
限流：用验证码来分散用户请求
禁止重复提交：一个用户秒杀后，把提交按钮置灰
动静分离：把静态数据直接缓存到离用户最近的地方，比如浏览器，服务器端的缓存
后端
限流：屏蔽无用的流量，允许少部分流量走后端
削峰：缓冲瞬时流量，尽量让服务器对资源进行平缓处理
异步：将同步请求转化为异步请求提高并发量
利用缓存：把商品信息放缓存中，减少数据库查询
负载均衡：用多个服务器并发处理请求，减少单个服务器压力
Kafka：高吞吐量的分布式发布订阅消息系统



网页爬虫设计：

用例:服务是抓取链接，生成包含搜索词的网页倒排索引，生成页面标题和摘要
用户输入搜索词可以看到相应结果列表
服务有高可用性
假设：搜索流量分布不均，有的搜索词热门有的冷门
用户很快能看到结果，能抓取 10 亿个链接，每月搜索量 1000 亿次

爬虫系统 URL 去重：用数组+hash 函数(求出 URL 的哈希值再对数组长度取模)，但是 hash 冲突时候就会误判(url 长度一样但内容不一样以为存在了),
减少哈希冲突：增长数组长度+用多个 hash 函数判断
布隆过滤器：判断某个元素存在的话可能不存在，但判断某个元素不存在则一定不存在
首先声明一个二进制数据，里面全是 0，然后把数据进行 n 次运算，每次运算都会找到一个位置，就把这些位置变成 1，在查询数据的时候也进行 n 次运算，如果一个位置为 0，那么我们就认为这个数据是不存在的；

哈希碰撞：两个不同的输入产生一个相同的 hash 值。

应用：分布式系统里的负载均衡和数据分布

哈希 2 种使用：

1.哈希取模：有 5 个服务器做负载均衡，然后我们就对请求的 ip 和用户 id 用哈希函数，然后把计算出的 hash 值对 5 取模，根据余数来分配到对应的服务器上
缺点：有状态的服务下，新增一个服务器会导致大多数请求要重新映射到别的节点，在 web 负载均衡的情况下，session 会存在每个节点里，如果增删节点会让几乎所有数据要迁移
2.一致性哈希：之前是用节点数量作为除数，一致性哈希的话是用 2^32 作为除数. 也就是 0-2^32 首尾连成一个环。先对服务器节点的 IP 哈希，然后除以 2^32，有请求来了，同样哈希然后除以 2^32, 然后的话是在环上顺时针找到第一个节点，这个节点就负责这个请求
优点：不会因为横向的伸缩导致数据大规模变动
缺点：在节点数量少的时候，会出现分布不均匀的情况(解决方案是在环上加虚拟节点)

哈希表特点：关键字在表中位置和它之间存在一种确定的关系

哈希冲突：把 key 哈希后的结果作为地址去放键值对的时候(hashmap)，但是地址上已经有别的键值对了

处理哈希冲突/溢出：

1.再散列法：哈希冲突时候，用探测技术在哈希表里得到一个探测序列。按照不同探测序列的形成方法可以分成 3 种：节点规模小时，再散列法节省空间

1.线性探查法：从已存的地址一个个往后找。

缺点：1.处理溢出需要另编程序 2.产生堆聚现象(存入哈希表的记录连成一片)，冲突可能性变大。

2.线性补偿法：探测的步长数要和表的长度互质 这样才能探测表中所有单元

3.随机探测：探测步长是随机数 这样能避免堆聚

2.拉链法：把哈希表变成指针数组，每个单元都是一个头指针，如果键值对有一样的散列地址就插到这个单元的链表里

优点：1.不容易有堆聚现象，平均查找长度短 2.装载因子大，节省空间 3.删节点简单因为不用像再散列法不能直接把节点置位空 因为会截断查找路径

缺点：指针要额外空间，节点规模小时，再散列法节省空间

哈希函数构造方法：

1.数字分析法：知道关键字集合的话，从中选出分布均匀的几位构成哈希地址

2.平均取中法：求关键字的平方值, 然后取平方值的中间几位作为哈希地址

3.分段叠加法：按哈希表地址位数把关键字分成位数相等的几部分然后相加，舍弃最高进位后的结果就是这个关键字的哈希地址

4.除留余数法：哈希函数 key 对小于等于哈希表长度的最大素数取模

版本控制：是记录软件开发过程里文件内容的变化用来查特定版本修订情况的系统

好处：

1. 记录什么人什么时候改了什么内容，每一次文件的改变，文件的版本号都将增加，确保在开发过程中由不同人所编辑的内容都得到更新

2. 方便并行开发，因为可以有效地解决版本的同步以及不同开发者之间的开发通信问题，并行开发中最常见的不同版本软件的(Bug 修正问题也可以通过版本控制中分支和合并的方法解决

版本控制系统：

Git(分布式) 客户端并不只提取最新版本的文件快照，而是把原始的代码仓库完整地镜像下来。这么的话，任何一个服务器发生故障，事后都可以用任何一个镜像出来的本地仓库恢复

SVN/ CVS(集中式) 一个单一的集中管理的中央服务器，它保存了所有文件的修订版本，然后一起工作的人都可以通过客户端连到这台服务器，拿到最新的文件或者提交更新

以下 3 个问题请求少的是影响不大，但请求多会造成服务器宕机，重启也没用

为了查询速度加快用缓存保存数据，让请求直接查缓存而不是数据库，缓存没有的话再查数据库然后写入缓存，缓存是有有效时长的，不然会一直占内存。

缓存雪崩：访问缓存的时候大部分缓存同时过期，请求就会打到数据库，如果请求数量足够大就会把数据库压垮

解决：1.不设置过期时间，缓存更新直接刷新 2.用集群把数据均匀分布在机器上
3.过期时间上加随机值防止缓存几种过期 4.用一些限流机制

缓存击穿：大量请求访问缓存里的一个热点 key，但这个 key 刚好过期了，大量请求就会穿过缓存直接到数据库

解决：1.让热点 key 不设置过期时间 2.设置定时任务把将要过期的 key 刷新
3. 在缓存中没有数据去数据库查询时加上锁，让一个线程去查询数据库以及更新缓存，其他线程等待，这样减小数据库压力

缓存穿透：大量请求访问缓存和数据库都没有的 key

解决：1.在缓存放查询的 key 并且把值设为 null，这样请求就不会打到数据库
2.在请求接口处检查，对于不合法的请求直接返回
3.用布隆过滤器(只有 0 和 1 的 bit 数组)-- 对待过滤的数值求 hash 散列后可以查看这个数组中对应的位置上是否为 1 来进行判断过滤

后端，提供客户端的数据请求和响应，客户端跟服务端保持连接、客户端发消息，首先是发给服务端、服务端将消息入库并将消息转发给对方；添加一个好友，是通过客户端操作的，搜索好友即向服务端请求搜索、服务端在“库”里面搜索结果、返回结果给客户端

内存和缓存：

内存是作为 CPU 和硬盘间的存储支撑，缓存是 cpu 的一部分，在 cpu 里，逻辑上在内存和 CPU 之间。cpu 读数据很快，内存就慢很多。缓存是为了解决 cpu 速度和内存速度差异问题的。内存里被 cpu 访问最频繁的数据和指令被复制到 cpu 的缓存里。在缓存没命中就去内存和磁盘找(数据库在磁盘里，缓存没命中的话数据库会把这些信息更新到缓存)

缓存的使用：

读数据：先读取缓存，若不存在则从 DB 中读取，并将结果写入到缓存中；下次数据读取时便可以直接从缓存中获取数据。

改数据：直接失效缓存数据，再修改 DB 内容

缓存不一致问题：

1.先删除缓存，数据库还没有更新成功，此时如果读取缓存，缓存不存在，去数据库中读取到的是旧值，就会缓存不一致

解决：延时双删--也就是为了避免更新数据库的时候，其他线程从缓存里读不到数据，就在更新完数据库后，再 Sleep 一段时间，然后再次删除缓存--sleep 时间大于读写缓存时间就行

ex：线程 1 删了缓存然后去更新数据库，线程 2 发现缓存被删了就去数据库读了旧值，并且它把旧值放回到了数据库，线程 1sleep 时间之后再去把这个缓存删了

1. 先更新数据库还没来得及删缓存，旧值就被读了--不一致

解决：用有监听 binlog(二进制日志文件)消息的消息队列(做核对的工作),也就是借助监听 binlog 的消息队列来删缓存,这样中间件直接帮忙做了解耦((解除 2 个东西互相影响的现象)不用单独引入一个消息队列)

为什么删除缓存不是更新缓存：这样会让缓存更新次数和数据库一样多，但缓存可能只被读了很少的次数

redis 数据结构：1.string(一个 key 对应一个 value) 2.hash(一个 mapmap)
3.链表(有序) 4.集合(无序) 5.zset(有序集合)